Bit Flipping Tutorial

An Uncomplicated Guide to Controlling MCU Functionality

Bit flipping—the process of flipping bits in registers—enables you to control a microcontroller's functionality. But what is the best method of bit flipping? Eric points you in the right direction with this helpful tutorial.

When a software engineer looks at a datasheet for a microprocessor or microcontroller, the first question that comes to mind is, How is this controlled? The processor's functionality is controlled through the writing and reading of registers at specific addresses in memory. Functionality is organized in the specific bits of the registers. The only way to control the functionality is by accessing the individual bits. This process is called bit flipping.

For most C language beginners, the best method of bit flipping is not always obvious. They usually choose bit fields. However, there are inherent problems with the way compilers may implement bit fields. Bit fields can be placed in a data type no smaller than an integer, which is larger than 1 byte in many cases. That won't work well for 8-bit registers. However, this greatly depends on your compiler and target processor. In addition to portability problems, bit fields cannot handle operating on multiple, incontiguous bits simultaneously.

A better solution is to learn more about C, which provides several operators that can be used to manipulate bits. The combination of these bit operators can do just about any kind of manipulation that an embedded programmer would ever need. These bit operators are inherently portable, unambiguous, and more flexible than alternative methods. C bit operators are absolutely essential for embedded software.

LOGIC & BIT OPERATORS

C language has a set of logic operators: AND (&&), OR (||), and NOT (!). These logic operators are used in flow control statements such as *if*, *while*, and *for* blocks. Truth tables show the rules of how the logic operators work. They list all the possible true and false values for the two operands and the corresponding result for the operator.

C language bit operators, at first glance, are similar to the logic operators. They include: AND (&), OR (|), NOT (~), Exclusive OR, or XOR, (^), Left Shift (<<), and Right Shift (>>). Bit operators follow the same rules as logic operators. The difference is that the former operate on bits within a byte instead of conditional expressions. When a bit is set (1), it is a true value in a truth table. When a bit is cleared (0), it is a false value in a truth table. Bit operators AND, OR, and NOT have truth tables that follow the corresponding logic operators (see Figure 1).

XOR means that if one—and only one—operand is true, then the result is true. Otherwise, the result is false. The XOR bit operator has no similar logic operator in C. The two shift operators (Left Shift and Right Shift) shift the bits in the variable to the left or right by the specified amount of bits. As long as the variable is an unsigned type, the new bits that are shifted are always set to 0.

Using these bit operators, you can manipulate bits to your heart's content. The next step is figuring out which bits to operate on.

BIT NUMBERS & MASKS

To access and operate on bits within a variable, you must be able to select the bit you want to operate on. Bits within a variable start at 0 for the least significant bit (LSB). With 8 bits in a byte, the bits are numbered 0 through 7.

A bit mask (either a variable or a constant) is used to select the bit to operate on. Any bit in the mask that is set represents the bits you want to operate on in the other operand. This effectively masks the bits you don't want to access, leaving them untouched. Only the bits you want to access can propagate through the operator. For example, if you want to operate on bit 0, your bit mask will be in binary

AND (&)

X	Y	Result
False	False	False
False	True	False
True	False	False
True	True	True

Or (|)

X	Y	Result
False	False	False
False	True	True
True	False	True
True	True	True

NOT (~)

X	Result
False	True
True	False

XOR (^)

X	Y	Result
False	False	False
False	True	True
True	False	True
True	True	False

Figure 1—*In truth tables for C bit operators, X and Y are single-bit operands within the variable.* Result *is a single bit within the variable as well. It shows the result of the bit operation.*

0000 0001. To access bit 7, your bit mask will be in binary 1000 0000. To access bit 5, your bit mask will be in binary 0010 0000.

PUTTING IT TOGETHER

To learn how all this works, let's do some exercises. Let's use a value of 0xF0 and see how the OR operator works on bit 0. Use the operator on each bit position in the value and in the bit mask to derive the result for the same bit position.

```
    1111 0000 (value 0xF0)
OR  0000 0001 (bit mask, bit 0 is set)
---------------
    1111 0001 (result)
```

Here, bit 7 of the value is ORed with bit 7 of the bit mask. True (1) OR false (0) results in true (1) for bit 7 of the result. Do the same for all bit positions. After all of the bits have been done, you can see that the result is similar to the original value, except that the bit selected by the bit mask (bit 0) is now set in the result.

Let's take the same value, 0xF0, and see how the AND operator works on bit 7. Again, use the operator on each bit position in the value and in the bit mask to derive the result for the same bit position.

```
    1111 0000 (value 0xF0)
AND 1000 0000 (bit mask, bit 7 is set)
---------------
    1000 0000 (result)
```

At the end, when you compare the result to the original value, you can see that the bits that weren't selected by the bit mask were cleared, and the bit that was selected remained its original value in the result. This operation can be used to read a bit in the original value.

You can combine bit operators to achieve different results. Let's take value 0xF0 and a bit mask with bit 7 set. But this time, use the NOT operator on the mask to invert all of the bits:

```
    1111 0000 (value 0xF0)
AND 0111 1111 (bit mask, inverse
               (NOT) of bit 7 set)
---------------
    0111 0000 (result)
```

At the end of operating on all the bits, the result looks just like the original value, except that bit 7 is now cleared.

Now let's try value 0xF0 with the XOR operator on a mask with bit 0 set:

```
    1111 0000 (value 0xF0)
XOR 0000 0001 (bit mask, bit 0 is set)
---------------
    1111 0001 (result)
```

The result looks like the original value, except bit 0 is now set. This seems similar to how the OR operator works. Or is it? Change the bit mask to have bit 7 set:

```
    1111 0000 (value 0xF0)
XOR 1000 0000 (bit mask, bit 7 is set)
---------------
    0111 0000 (result)
```

Now the result looks like the original value, but bit 7 is clear. The XOR operator will flip, or toggle, the bit in the value no matter what the original state of that bit was. You can write all of these operations in C.

Setting a bit requires the OR operator. Read the value in a variable, OR the value with the bit mask, and assign it back to the variable:

```
var = var | mask;
```

You can combine the assignment operator with the bit OR operator in a compound assignment:

```
var |= mask;
```

Clearing a bit requires the NOT and AND operators. Read the value in a variable, AND the value with the NOT of the mask, and assign it back to the variable:

```
var = var & ~mask;
```

Again, you can use a compound assignment:

```
var &= ~mask;
```

The NOT operator is used to invert the mask so you can use the same mask to either set or clear a bit or to perform other operations. The meaning of the mask stays the same: the bits that are set are the bits you want to operate on in the variable. If the mask is a constant, then the compiler will automatically perform the NOT operation and it won't be done at runtime.

Toggling a bit requires the XOR operator. Read the value in a variable, XOR the value with the mask, and assign it back to the variable. Again, you can use a compound assignment:

```
var ^= mask;
```

Reading a bit requires just the use of the AND operator. Read the value in a variable and AND the value with the bit mask. If the bit is clear, the result is a zero, which is interpreted as a logic false. If the bit is set, the result is the value of the mask, which is a non-zero (not a one). The non-zero is interpreted as a logic true. For example, to read a bit and check to see if it is set:

```
if(var & mask)
{
   // bit is set
}
```

To check to see if a bit is clear, check if a bit is set (as above), but use the logic NOT operator with it. Be sure to use parentheses to avoid ambiguity. For example:

```
if(!(var & mask))
{
  // bit is clear
}
```

READABILITY & SIMPLIFICATION

It can be difficult to remember the various bit operators. Things can get even more complicated when you're using many in a given expression. C preprocessor macros can help simplify expressions and improve readability. Let's start by generating a bit mask.

There is a simple relation between a bit number and a bit mask. Take the value of one and shift the bits over to the left by the bit number. The value of the new bits shifted in on the right will be all zeros. You can define a

Listing 1—*C preprocessor macros can help make the common bit operations more readable.*

```
#define bit_set(v, m)      ((v) |= (m))
#define bit_clear(v, m)    ((v) &= ~(m))
#define bit_toggle(v, m)   ((v) ^= (m))
#define bit_read(v, m)     ((v) & (m))
```

macro to implement the following:

```
#define bit(num) (1 << num)
```

This macro takes a bit number as its argument and uses that bit number as the number of times it should Left Shift a constant of 1. To represent bit 1 in a bit mask, use `bit(1)`, which means that the constant value of 1 is shifted to the left one time. In binary, this would cause 0000 0001 to become 0000 0010 (in hex, 0x01 would become 0x02). If the bit number is 0, then the constant 1 will be shifted to the left 0 times, leaving the result as 1.

You can also use macros for setting, clearing, toggling, and reading bits (see Listing 1). Each macro accepts a variable as the first parameter and a bit mask as the second parameter. There is a good reason why you don't want these macros to accept a bit number but you do want them to accept a bit mask: defining multiple bits. If the macros accept only a bit number, then they can operate on only one bit at a time. This is acceptable for most situations. However, there are times when you want to operate on multiple bits simultaneously. In those cases, it's relatively easy to do. Use the `bit(num)` macro to convert the bit number to a bit mask. Then, OR all of the resulting bit masks together to get a composite bit mask representing the bits you want to operate on. For example:

```
bit_set(PORTD, bit(0) | bit(1)
  | bit(2));
```

Another advantage with this type of implementation is that you can operate on multiple, incontiguous bits simultaneously:

```
bit_set(PORTD, bit(0) | bit(3)
  | bit(5));
```

HAL

Putting together macros to help you operate on bits is just a means to an end. One of the most important things you'll want to be able to do is easily model the microprocessor you're using. You'll want to create a Hardware Abstraction Layer (HAL) that will help you control the processor.

For example, in an Atmel ATmega8 AVR, you don't really care what it means to set bit 7 in I/O address 0x0A. But you do care about enabling the receive interrupt for the USART. The "magic numbers" of I/O addresses and bit numbers are just a way to achieve the real goal of controlling the device and its subsystems.

You can easily implement this, again by defining a set of preprocessor macros (see Listing 2). Writing and reading code is made easier because it's a lot easier to remember what action you want to perform on an object (e.g., enable the USART receive interrupt) than it is to remember what register and bit is associated with that action. Listing 3 demonstrates the kind of code you can write.

Sometimes there are operations on the processor registers that don't easily map to the simple `bit_set()` and `bit_clear()` macros. One example is the ATmega128 processor. Bits 0 through 2 are labeled ADPS0, ADPS1, and ADPS2, respectively, in the ATmega128's ADC control and status register A (ADCSRA). These bits are used to select the ADC prescaler and should be used as a single value.

To change this value without disturbing the other bits in the register, read the register and clear out the bits using a bit mask comprised of all of the bits in question. Then, set the bits using a bit mask comprised of the value that you want to set. You may want to further mask the value you plan to set with the mask of just the bits you intend to operate on, limiting the value to the correct range. And lastly, assign the entire thing back to the register. Believe it or not, this can be done in one line of C and assigned to an easy-to-read

Listing 2—*C preprocessor macros with meaningful names can transform bit flipping into a Hardware Abstraction Layer (HAL) that you can reuse in other applications and with other processors within the same family.*

```
#define usart_receive_interrupt_enable()    bit_set(UCSRB, bit(7))
#define usart_receive_interrupt_disable()   bit_clear(UCSRB, bit(7))
#define usart_transmit_interrupt_enable()   bit_set(UCSRB, bit(6))
#define usart_transmit_interrupt_disable()  bit_clear(UCSRB, bit(6))
```

Listing 3—*You can use an HAL to initialize a USART. The HAL hides the bit flipping that is done to control the processor.*

```
void usart_init(void)
{
        usart_receiver_disable();
        usart_transmitter_disable();
        usart_baud_rate(9600UL);
        usart_receive_interrupt_enable();
        usart_transmit_interrupt_enable();
        usart_receiver_enable();
        usart_transmitter_enable();
        return;
}
```

macro in the HAL:

```
#define adc_prescaler(v) \
ADCSRA = (ADCSRA & ~(bit(0)| \
bit(1)|bit(2))) | (v & (bit(0)| \
bit(1)|bit(2)));
```

You can write a HAL for each subsystem on the device, such as the UART, ADC, or SPI bus. In reality, you will probably have to create functions as well as macros to create a complete interface to a subsystem. But after the HAL or interface is completed, you can reuse it in future projects that feature that device.

API

Now that you have a HAL that basically models the microprocessor, you can take things one step further. You can add a layer that is specific to the application, the Application Programming Interface (API). This API is how your actual application will be implemented on the device hardware. The API will use the definitions that were created for the HAL.

Consider Listing 4. There are three macros that are specific to the application you're writing: `motor_status()`, `status_led_on()`, and `status_led_off()`. The latter two are implemented as a simple clear or set to the port pin where the LED is connected. Macro `motor_status()` is defined as the value of the external interrupt flag 1. This, in turn, is defined as specific bits in a specific register. What makes this scheme advantageous is that it is now

Listing 4—*You can create C preprocessor macros to implement application-specific functionality that can use either the HAL or lower-level bit flipping to control the hardware.*

```
// HAL definitions (which can be in a separate header file).
#define external_interrupt_1_flag()  bit_read(GIFR, bit(7))
#define external_interrupt_0_flag()  bit_read(GIFR, bit(6))
// API
#define motor_status() external_interrupt_1_flag()
#define status_led_on()       bit_set(PORTA, bit(0))
#define status_led_off()      bit_clear(PORTA, bit(0))

void foo_task(void)
{
        if(motor_status())
        {
                status_led_on();
        }
        else
        {
                status_led_off();
        }
        return;
}
```

easy to redefine how things are hooked up. What if you had to change the reading of the motor status from the external interrupt 1 flag to the external interrupt 0 flag? It's easier to make the change in the definition of `motor_status()`, like so:

```
#define motor_status() \
external_interrupt_0_flag()
```

The great thing is that you have to change it in only one place in the code. Then the entire application will use the new definition. It's also easier to remember the HAL definition of external_interrupt_0_flag() than it is to remember the exact register, bit number, and bit polarity that corresponds to reading the external interrupt 0 flag.

Note that `status_led_on()` is defined as setting a bit. And `status_led_off()` is defined as clearing a bit. If you need to change the hardware to have the processor sink the current of the connected LED (thereby changing the polarity of turning on and off the LED), then it would be easy to change the definition of `status_led_on()` to clear the bit and `status_led_off()` to set the bit. Again, you would have to change the definitions in just one place in the code.

If the API is implemented in such a way that the function and macro names contain the name of the subsystem/device and the action being performed with that device (i.e., an object and a verb), then the code is often self-documenting. At the very least, it's easy to read and follow what is happening.

GETTING CONTROL

An embedded software engineer controls a microprocessor by flipping bits in registers. Learning how to effectively manipulate bits in C language is a basic skill in writing embedded software. The most flexible way to do this is by using the C language bit operators. You can create C preprocessor macros to simplify the common usage of the bit operators.

You can expand on this to create a layer of macros and/or functions that can model the processor with names that are symbolic of the processor's functionality. This creates a HAL.

Ultimately, you can create a layer of macros or functions that implement your application functionality: an API. The API is used in the actual implementation of your application. This segmenting of functionality provides the greatest flexibility because it allows easy change of the software, especially when the underlying hardware is changed.

Eric Weddington has been a software engineer for 14 years and has written embedded software for the last 10 years. He is the Senior Software Engineer at a company that makes RFID products for the pharmaceutical laboratory industry. He can be found at an altitude of 8,000 feet in the foothills west of Denver, Colorado. You may reach Eric at arcanum@users.sourceforge.net.

SOURCE

ATmega8/128 processors
Atmel Corp.
www.atmel.com